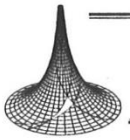

7/1 (2009), 101–117
tmcs@math.klte.hu
http://tmcs.math.klte.hu

Teaching
Mathematics and
Computer Science

Teaching Sorting in ICT

Péter Szlávi and Gábor Törley

Abstract. This article is aimed at considering how an algorithmic problem – more precisely a sorting problem – can be used in an informatics class in primary and secondary education to make students mobilize the largest possible amount of their intellectual skills in the problem solving process. We will be outlining a method which essentially forces students to utilize their mathematical knowledge besides algorithmization in order to provide an efficient solution. What is more, they are expected to use efficiently a tool that has so far not been associated with creative thinking. Sorting is meant to be just an example, through which our thoughts can easily be demonstrated, but – of course the method of education outlined can be linked to several other algorithmic problems, as well.

Key words and phrases: algorithm, Excel, algorithmic thinking, sorting, program efficiency, combinatorics, permutations.

ZDM Subject Classification: C74-76, K24-26, Q34-36, R24-26, R74-76.

1. Introduction

Informatics as a school subject includes teaching algorithmic thinking, or even teaching programming, as well as the most widespread application systems like spreadsheets. [15] We assume that sorting can be introduced as a fruitful topic in ICT teaching, which is what this article aims to discuss.

Sorting algorithms have an advantageous feature: namely, they may set a challenge for both excellent and not so excellent programmers since the problem itself can be formed in an easily understandable way, and there exist an obvious algorithmic solution to it. If students are given a task that is based on any of the

sorting algorithms and they have to approach the solution comprehensively, not only their algorithmic skills, but some "extra" features will also be utilized.

The well-known PISA assessment [9] has shown that there are few practical, thought-provoking and mind-developing elements in today's Hungarian education. It will be clear that a varied approach to sorting – through algorithmization, experimenting, using spreadsheets – can efficiently develop creative and practical thinking.

1.1. Sorting Task

What is a sorting task (ST)? Arrange the elements of a sequence in such a way that each member comes before every other member that is greater than it from a previously defined aspect, which must be, of course, of such nature that it can establish an order i.e. which of the two elements is smaller. If the element is complex, the sorting "aspect" is often embodied by an element-like[1] field or perhaps a function that is defined on the type of elements and returns some element-like value.

1.2. Sorting is the Go of algorithmization

Sorting is a good "training field" regarding algorithmization, as sorting tasks can easily be grasped even by beginners, and what is more, common sense can help to find a correct solution. On the other hand, more advanced students can also find it challenging when they look for a more efficient solution. Therefore, it may provide students with various programming experience with thought-provoking topics, just like the popular Japanese board game, the Go.

1.3. Varietas delectat

The peculiarity of the ST can also be approached from a point of view that it is sensible to consider solution variants. There is not *one single optimal* solution. It is regarded as rarity in the world of basic algorithms, which secondary school students encounter. Just remember when algorithms are taught, *programming theorems* [14] are just based on the idea that a *task* captured in a certain abstract

[1]Elementary types are pre-considered ordered, i.e. the question of "order" regarding them can be answered.

way can have – more or less – *one* unambiguous abstract *solution*. Their "togetherness" can be proved with formal tools [3], [13], although it is indifferent from the point of view of secondary education.

There are two essential reasons for working out *variants*: 1) Following different "algorithmic philosophies" (concepts, ideas), different correct solutions can be found. [8], [7]. 2) As for efficiency, variants behave differently. It is especially striking when efficiency is defined generally. In [17] the author adds the dimension of complexity to the two well-known "dimensions" of efficiency: speed (i.e. time dimension, also known as time complexity) and memory space requirement (i.e. space dimension, also known as space complexity). This new dimension is used to indicate how much mental energy the program designer requires for comprehending the algorithm. We may find any quantitative measures of psychological complexity, or any degrees appearing in the literature (the cyclomatic complexity of a program, depth complexity, structural complexity, etc.). The application of a thus generalized measure of efficiency makes it possible for the simple sorting algorithms to become the "competitors" of even the quickest.

There is one additional educational benefit, similar to which students cannot have met before while writing programs: namely, *it is impossible to* unambiguously decide which one is better out of a pair of algorithms belonging to a task. They must *discuss* the answer just like in mathematics at secondary school.[2] It follows from the above, that the goodness of an algorithm cannot be measured by one single number, but one needs (at least) three.

Now it can be stated that the well-known Latin proverb of the heading "variety is the spice of life" holds for applying ST in teaching algorithms as well.

2. Sorting in a nutshell

Below a few sentences will be dedicated to the essence and algorithm of sorts that students will probably apply. (Algorithms are now confined only to the body of the sorting procedure without its heading and declarations in a Pascal-like language. The `Swap` procedure, which is not detailed below, swaps the values of two variables given as parameters.)

The principle of *Simple changing sort* is that if the correlating order of the couples in the comparison is wrong, they will be exchanged, i.e. it is a so-called

[2]Just think of e.g. the solution of the quadratic equation. When doing it, students make a similar methodological discovery: but in the field of mathematics.

transposition sort. It is not too efficient as it contains too many superfluous changes.

```
For i:=1 to N-1 do Begin
  For j:=i+1 to N do Begin
    If X[i]>X[j] then
      Swap(X[i],X[j])
  End;
End;
```

Figure 1. Simple changing sort

As for its philosophy, selection *sort* is *selective:* it always selects the next sorted element (selection), and then swaps it into its position. Its operating principle is not too sophisticated: first find the lowest element of the array, and then swap it with the value in first position. This way that element gets to its position. Then the same procedure is applied for elements $2 \ldots N$, too, when the second element will get to its position in the sorted sequence, etc.

```
For i:=1 to N-1 do Begin
  mini:=i;
  For j:=i+1 to N do Begin
    If X[mini]>X[j] then
      mini:=j
  End;
  Swap(X[i],X[mini])
End;
```

Figure 2. Selection sort

The basic principle of *bubble sort* is interchanging adjacent elements. In the first pass starting from the end of the array, every item is compared to its left neighbour. If they are in the wrong order, they will be swapped. At the end of the first pass, the lowest item will surely get to its sorted position. In each next pass we will start from the end of the array, but we will need fewer and fewer comparisons as the beginning of the array is gradually becoming ordered. This is *transposition* sort.

The operation of *Insertion sort* mostly resembles to picking up cards from a table and putting them to their places. Take the next element, and find its position in the already ordered subsequence to the left. As for it philosophy, it is an *insertion sort.* Its principle: putting the next element to the right sorted position in an already ordered subsequence.


```
i:=N;
While i≥2 do Begin
  swP:=0;
  For j:=1 to i-1 do Begin
    If X[j]>X[j+1] then Begin
      Swap(X[j],X[j+1]);
      swP:=j
    End;
  End;
  i:=swP
End;
```

Figure 3. Improved bubble sort

Shell sort is not an independent method, but it can be used together with several sorting algorithms described above. The idea is that it greatly improves sorting by comparing elements separated by a gap of several positions first, which lets an element take "bigger steps" towards its expected position, thus making the original method more efficient. This concept works extremely well in insertion sort. [14], [8], [7]

3. The description of sorting

This chapter is dedicated to the structure and analysis of complex tasks based on a sorting algorithm. To put it shortly: find one or more formulas to the chosen (or possibly several) sorting algorithm(s) that well characterize(s) its/their efficiency. Specifying the task is an important part of the solution; it is, eminently, what is meant by the efficiency of an algorithm. It will be clarified below.

3.1. Characteristic features – in the language of mathematics

Special attention will be paid on the time behaviour of algorithms. Such feature must be found that expresses a real operation. It is not a good idea to choose, e.g., – the obvious – real running time, as it also depends on hardware, moreover, on its momentary resources. Worse results will be measured on a slower computer with less memory. If a sorting program is tested on several computers of the same capacity, – at first sight – astoundingly, the time results will differ even for the same inputs. Just remember that there may start, e.g., a virus check in the background, which often happens if one uses today's "multitasking" operating systems, and immediately a new value is added to the time of sorting,

which is absolutely independent of it. Of course, the same way it might happen that one single computer returns different results for the same inputs at different times, which is also unacceptable. If the above mentioned anomalies caused by parallel running were set aside, the actual execution time would not be acceptable, either. As it is obvious that running time sharply increases with the increase in the "complexity" of the data to be sorted (e.g. the items of the sequence are records), since more time is required to move more extensive data, although the essence, the algorithm itself has not changed at all.

It follows that one needs some more *general features* like the two most typical algorithmic operations: the number of a *comparisons* and the number of *moves*. These parameters can really describe the "substantive" time efficiency of an algorithm. So one can exclude any other "decorations" of an algorithm that ornate the various algorithm variants more or less the same way. Accordingly, these parameters are suitable for comparing various sorting algorithms.

It is easy to define the two *extreme cases* – the maximum and the minimum running time: after determining the two inputs causing extreme runnings, one must only follow the double cycle constituting the algorithm, while the execution of the two operations important to us is counted. Note that it is not true in each and every case that it would be easy to determine the input belonging to the extreme running time of the given algorithm. In fact, the teacher must call students' attention this very fact. This calculation does not require "higher academic knowledge"; one must only recognize the applicability of the sum of arithmetic sequence so students' mathematical knowledge of the secondary school level is enough, but – fortunately! – absolute necessity. In the tables below, the above mentioned features of algorithmized procedures are summed up, completed with asymptotic notations common in mathematics. [1], [2], [5]

Aspect	Min	Max	
Number of comparisons	$N(N-1)/2$		$\Theta(N^2)$
Number of moves	0	$3N(N-1)/2$	$\Theta(N^2)$

Figure 4. Features of simple changing sort

Aspect	Min	Max	
Number of comparisons	$N(N-1)/2$		$\Theta(N^2)$
Number of moves	$3(N-1)$		$\Theta(N)$

Figure 5. Features of selection sort

Aspect	Min	Max	
Number of comparisons	$N-1$	$N(N-1)/2$	$\Omega(N), O(N^2)$
Number of moves	0	$3N(N-1)/2$	$O(N^2)$

Figure 6. Features of improved bubble sort

3.2. Let us examine a "typical case"!

Now the two extreme values, i.e. the best one and the worst one, are known. Obviously, they occur fairly rarely. Therefore, some questions may arise: What should be considered a "typical case"? How many operations does a typical case require? Can the average of the best and worst cases be taken as the operation requirement of a typical case? Unfortunately, it cannot, as a 4-item task below clearly shows it.

To define a typical case, one will need the concepts of *inversion* and *number of inversions*. (In class, of course, it is not necessary to state a precise definition and, thus, perhaps frighten students off further thinking. It is absolutely satisfactory to refer to the essence with examples.) Consider each sequence an N element permutation. Thus N different elements are given. Take a permutation of these N elements (e.g. the ordered one), and consider it the *natural order*. If two elements are examined in a permutation, it can be stated which element comes before the other one. Call it the relationship of the two elements. Say that the two elements are in *inversion* if their relationship differs in the examined permutation and in natural order. The number of couples in inversion is called inversion number. [7], [4] Then examine all permutations of a sequence, and take the *distribution* and *average* of inversion numbers. (Of course, instead of the precise concept of distribution, refer to the easily conceivable, comprehensible counterpart, the frequency histogram.) Why does it seem to be a better approach than the average of the best and worst cases? It can be explained by the fact that the smaller the inversion number of a sequence, the more ordered it is and the fewer elements

are in a "wrong" position, so the fewer steps and comparisons will be necessary to make the sequence sorted. That is to say, supposing an arbitrary sequence might occur with the same chance, one will get the average inversion number after performing the inversion calculations for all sequences.

Let us see a simple example for $N = 4$! Let us apply, for instance, *improved bubble* sort! The number of comparisons of the algorithm is $N-1$ in the best case, presently 3; and in the worst case $N(N-1)/2$, that is 6; its number of moves is 0 in the best case, and $3N(N-1)/2$ in the worst case, that is 18. (See Figure 6.) The average of minimums and maximums for the number of comparisons is 4.5, and for the number of moves is 9.

Let us see what result one will have by counting the inversions! The number of all possible sequences with 4 different terms is 4! that is 24. In the table below we will follow the possible cases for $(1, 2, 3, 4)$ example sequence:

Permutation	*Inversion*	*Permutation*	*Inversion*	*Permutation*	*Inversion*	*Permutation*	*Inversion*
1, 2, 3, 4	0	2, 1, 3, 4	1	3, 1, 2, 4	2	4, 1, 2, 3	3
1, 2, 4, 3	1	2, 1, 4, 3	2	3, 1, 4, 2	3	4, 1, 3, 2	4
1, 3, 2, 4	1	2, 3, 1,4	2	3, 2, 1, 4	3	4, 2, 1, 3	4
1, 3, 4, 2	2	2, 3, 4, 1	3	3, 2, 4, 1	4	4, 2, 3, 1	5
1, 4, 2, 3	2	2, 4, 1, 3	3	3, 4, 1, 2	4	4, 3, 1, 2	5
1, 4, 3, 2	3	2, 4, 3, 1	4	3, 4, 2, 1	5	4, 3, 2, 1	6

Figure 7. List of inversions for certain permutations $(1, 2, 3, 4)$ in natural order

The frequency of inversions will be as follows:

Inversions number	*Frequency*	*Inversion number*	*Frequency*
0	1	4	5
1	3	5	3
2	5	6	1
3	6		

Figure 8. The relative frequencies of inversion, in $(1, 2, 3, 4)$ natural order

The sum of inversions is 72, their average is $3(= 72/24)$, in other words: *the average number of inversions is the inversion number weighted with the relative frequency of inversions.*

The above frequency table shows that it is not reasonable to estimate the average running time with the average of the minimum and maximum, as the relative frequency of these extreme permutations – and thus their role affecting general behaviour – will dramatically decrease with the increase of N.

Unfortunately, it can be seen that this level of knowledge is already beyond secondary school mathematics. Does that mean that we will have to renounce the formula of "general behaviour", the most expressive comparison of algorithms?

3.3. Two ways – two subtasks

Now the more complete task that is based on sorting algorithms can be formed this way: *compare the sorting algorithms by their time efficiency: by giving the expected minimum, maximum and average numbers of comparisons and moves: (MinH(N), AveH(N), MaxH(N)); MinM(N), AveM(N), MaxM(N))).*

Let us now return to the question raised at the end of the previous chapter: Should we renounce the formula of "average behaviour"? Definitely, not! Only the approach must be changed. Let us rely on the students' existing knowledge. They can algorithmize to a certain extent, and so can they use a spreadsheet. The concepts of inversion and frequency can be introduced, but word them in such a simple way that a secondary school student can understand them. What do we want to know?

The plan for proceeding: we are going to draft two subtasks related to the original question, to which a program should be written, respectively. These programs will produce outputs only to some small parameter values, and we will try to set up the required formulas relying on the results. When making the formulas, we will intensively use the usual services of spreadsheets.

With the first program, we will tabulate inversion frequency to certain N, which will then help us find a formula to the average inversion number (see Figure 9.). This application still displays the values of factorial, the average and sum of inversion and that of the greatest inversion to N. By intuition, the latter values represent important information (the sought function might depend on N through one of them), and it does not pose too much difficulty for the program to define them. For a programmer it is nothing but "a piece of cake". It also writes out the displayed values in a file so that it could also be processed with a spreadsheet.

When using the program, one will be sad to see that after certain N, running becomes unbearably slow. Finding the formula will have to be based on the so far generated frequency sequences. It is at this point that we will turn to spreadsheets for help for the first time. Now we will have to notice a connection hidden in a heap of numbers of a table filled from the generated file. We will try pot-luck to find this connection. A spreadsheet specifically suits this empiric approach.

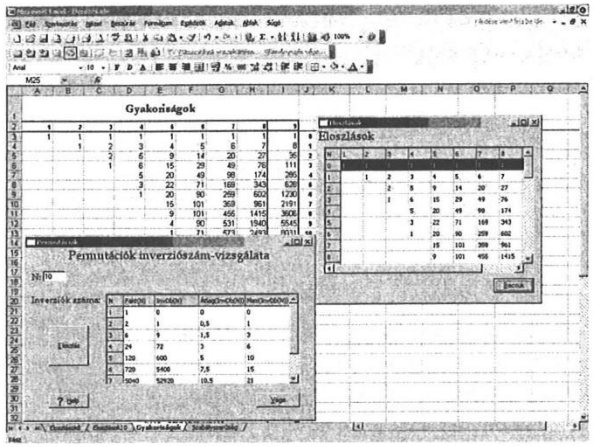

Figure 9. The program counting inversion frequency with a table in the background in which we hunt for connections. (In this case let us write the supposed formula in column I.)

Figure 10 shows the surprising connection with which the inversion distribution for $N = 9$ was determined from the data generated to $N = 8$.

Our trials were immensely naive, but in the end they proved to be practical. We only made the sum and difference of the values of the previous row and column in order to generate the value of a given cell, whereupon the following connections were outlined. Let `freq` be an $m \times n$ matrix (`freq`$\in \mathbb{N}^{m \times n}$), where n is the value up to which the frequency of the inversion is examined, and m the maximum inversion number of a sequence with n elements ($= n(n-1)/2$). n is indexed starting from 1, whereas m from 0. Discoveries:

1. `freq(0,i)=1`,
2. `freq(1,i)=i-1`,
3. `freq(j,i)= freq(j-1,i)+freq(j,i-1)`, `j=2..i-1` – that is the sum of elements one to the top and one to the left;
4. `freq(j,i)=freq(j-1,i)+freq(j,i-1)-freq(j-i,i-1)`, `j=i..i*(i-1)/2` – that is the previous formula has slightly been modified: the element in the column to the left of `freq(j,i)`, with i rows above is subtracted from it.[3]

[3]See: [7] pp. 28–29

Figure 10 clearly shows that this way distribution $N = 9$ could have been determined from distribution $N = 8$, and the checks have confirmed the correct operation of the formula for the previous cases.

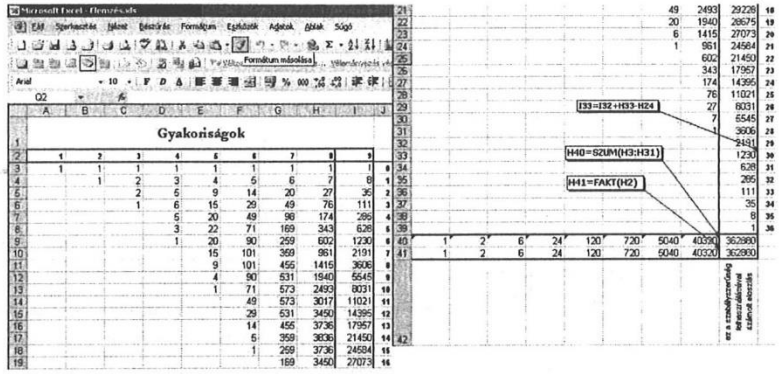

Figure 10. The table "processing" the data of the program counting inversion frequency, and the surprising connection in column I.

Having set up such a formula, we have reached a fine educational opportunity. We have now the chance to lead our students open to mathematical thinking into further activities unknown in traditional programming, as they are trying to prove the 3. and 4. formulas. The proving in fact can be done following basic considerations, as recursion automatically brings up the guiding principle. Let us see, as an example, the proof of the 3. formula:

`freq(j,i)` indicates those series which contain j inversions. The first member of the formula (`freq(j-1,i)`) remounts to those series of the length of i which have one less inversion, while the other (`freq(j,i-1)`) operates with the one shorter. From the latter, it can be suspected that it has something to do with a special element fixed in a special place. Indeed it does: let us now divide in two the series of the length of i. Into one of them, let us put those whose last (that is, the i) element is of the biggest value, while into the other all the rest goes. Consequently, in the two groups the total of the series will be `freq(j,i)`, and, obviously, in one `freq(j,i-1)` and in the other `freq(j-1,i)`. The number of those series of the length of i containing j inversions, which have the biggest in the end, will be `freq(j,i-1)`. It is so because in relation to the others the biggest is positioned correctly, that is, it does not increase the number of inversions, so it is the preceding `i-1` element which is responsible for the j inversion. The bringing

out of the other element, however, requires a trickier idea. What should be played on is that the biggest element is not in its place. In such a case, it can be moved one backward, and like this it will be in a "wrong" position relative to one less element, decreasing the number of inversions with one. Fulfilling this with each series of the first group, we lessen the number of inversions of each. In this way, we have brought their number to the number of the series with one less inversion (`freq(j-1,i)`), justifying the 3. connection. The other proof is more complicated and lengthy, so we set it aside now.

The connection found this way then must be adjusted to the algorithms, i.e. the inversion number, the number of comparisons, and the number of moves can be related only by considering the specific algorithms. No matter how painful it is, this relationship cannot be easily set up. The problem is that it is not enough to have the inversion number, but it is just as important – it is easy to see – to know the whereabouts of the inversion.[4] This way there cannot be found such a formula in which there are only terms depending on the inversion number. So this way we cannot come closer to our original goal. Indisputably, however, this little sideline has been useful to make us carefully consider: efficiency indices are strongly related to the features of permutations, the inversion number, and the whereabouts of inversions. Now let us return to our original aim and choose a different approach!

The second program (see Figure 11) determines the average empirically, i.e. it generates all the possible permutations for N, and then computes the best and worst cases, the average numbers of comparisons and moves one by one by sorting algorithms. It can be seen that as N increases, the program will become unbearably slow again. The continuation is just like before: connection analysis using the so far generated data with the help of a spreadsheet.

The generated data are gathered in a table again to find connections among them. The table has been divided into three ranges. The values generated by the program were loaded into the first one: the minimum, average and maximum values of comparisons and moves belonging to the examined sorting, for a certain N. The second part of the table is the "field" for experimentation. The supposed

[4]A new sub-task may be set that pries into this question. Namely, is it true that permutations with the identical inversion numbers cannot necessarily be sorted with identical numbers of comparisons and moves. Idea: with a little change in the previous program, one can have a solution that will write all the characteristic features of a permutation related to sorting in one file: the permutation itself, the inversion number, the number of comparisons, and the number of moves, then reading them in a spreadsheet and sorting them by inversion number, the above question can be answered on the face of it.

Figure 11. An output of a program characterizing sorting empirically.

formula depending on N was entered here, and here appears our predicted value belonging to a given N. Whereas the third part of the table shows the difference between the previous two coherent values. (See Figure 12.) Our idea on how to use the spreadsheet is to vary the formulas in the middle part of the table until – according to our expectations – the part of the table of difference contains only 0 values.

We are particularly interested in the formulas of averages, since – as it was claimed before – students are able to set up formulas belonging to minimums and maximums using elementary methods. It is obvious that we cannot aim at introducing the world of numerical methods to students. Relying on the formulas received for the maximum, students foresee that the average will resemble to some quadratic formula, which seems credible on the "structure" of algorithms: every sorting algorithm is essentially a double loop. When searching for a formula, we started from the formula determining the maximum, because that seemed most expressive. This formula was varied by multiplying it with a constant, and/or adding a constant value. Note that the table contains the current value of N in column A of the given row, thus a connection more sophisticated than the linear transformation regarding the maximum can be constructed. Figure 13 shows[5] that

[5]Background colour and automatic formatting have been used to highlight the difference acceptably close to 0 (absolute difference $\leq 0,5$), so that the values that are somehow problematic can stick out.

with simple algorithms there was no or little difference. With more complicated ones, however, the greater N was, the farther the value of the formula got from the real average, but these differences were less than 1% in proportion (= difference / expected value), thus, they give a reasonable picture about the development of the real average for the examined N. Presumably, they describe well the average asymptotically, too.

The above "two ways" example show that a fairly good picture has been received about the general behaviour of our algorithms, after turning to a spreadsheet for help, through experimentations and with some creativity.

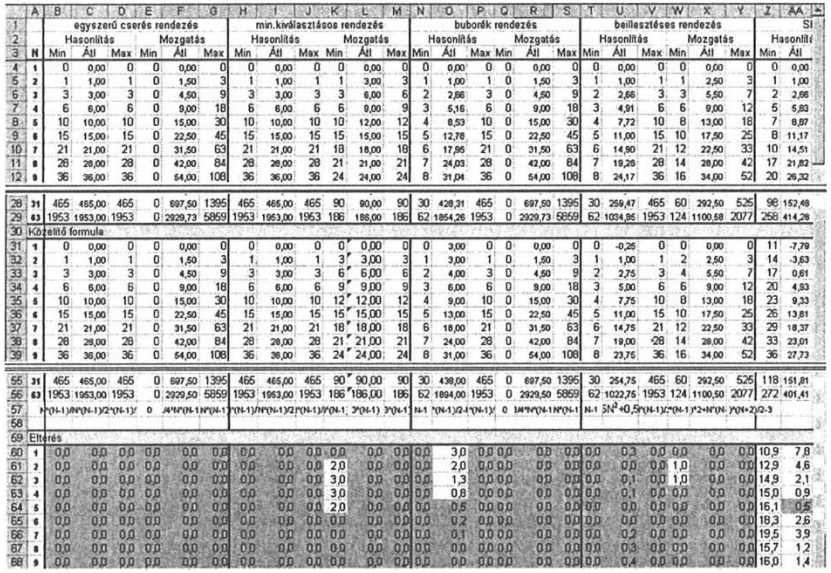

Figure 12. The spreadsheet examination of the output of a program empirically characterizing sorting – table structure.

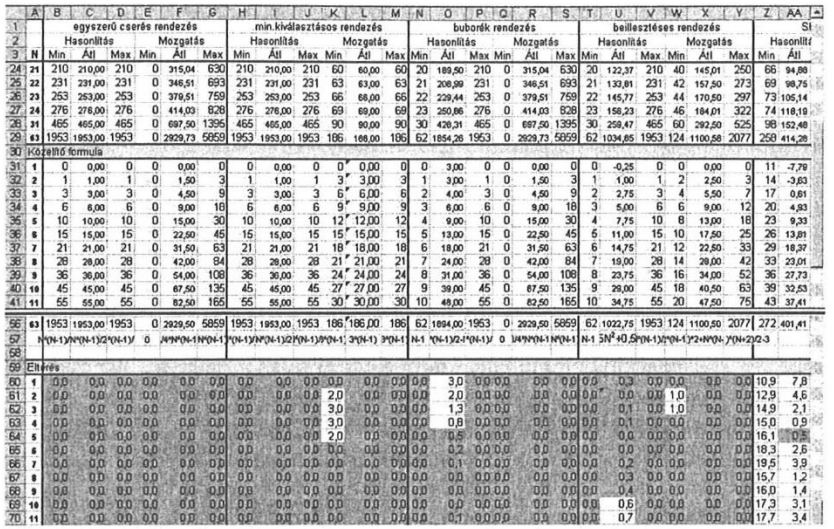

Figure 13. The spreadsheet examination of the output of a program empirically characterizing sorting.

4. Summary

In this article sorting as a programming task has been examined from the point of view of how to use it in ICT classes. It has been stated that classroom use may have several advantages. It can be seen that regarding the making of algorithms, it poses worthwhile problems for students of all levels, and it enables them to further refine their algorithmic knowledge. Above it has just been referred to that those who are already over elementary sorting will obviously find real challenge in trying out or individually discovering some more "professional" documented sort (e.g. Shell sort, Quicksort, Heapsort). It should be added that when writing the program of the task mentioned secondly, students will have to tackle reasonably an extra challenge that comes from the increase of the number of permutations when N is greater. Obviously, only experienced programmers will be able to overcome this problem.

In order for students to be able to solve the complex task, they will have to become familiar with the necessary algorithm(s). What is more, without an accurate analysis of algorithms, they will not reach their goal. It is important

to stress: in our concept the main emphasis is given to the algorithm or the main principle of the algorithm, the profound acquisition of which is an explicit objective.

Another characteristics of the problem-solving strategy is that it requires students to apply an all-round strategy. Besides the algorithmization of the basic problem, they should make fundamental, mathematical considerations. The way to reaching the objective goes through solving some auxiliary problems, which will require using certain application systems, e.g., a spreadsheet and perhaps – writing some simple programs that can usually be traced back to patterns.

Consequently, students will develop both their *algorithm-creating* and their *mathematical knowledge*, moreover, they will mobilize their *creativity*, experimenting skills, and by making them use several tools to solve a problem, their general knowledge will also become more complex and "accessible."

Finally, we claim that problems can be formulated in this way not only to the above examined sorting algorithms, but it can well be used with problems belonging to other algorithm classes. For instance, problems that are connected to graphs [8], [1], [16], [18] or can be solved with the help of dynamic programming [11], [12], backtrack [14] or greedy algorithm [10] can provide an obvious starting point for the above method, since "formally" there are – one can say – solution variants to such problems and the above described efficiency analysis approach is raised by nature if one wants to choose one out of them. We would not like, however, to claim that mathematical and/or spreadsheet knowledge can only be applied for the sake of efficiency analysis of an algorithm.

References

[1] T. Cormen, C. Lieserson, R. Rivest and C. Stein, *Introduction to Algorithms*, MIT Press and McGraw-Hill, 2001.

[2] R. Graham, D. Knuth and O. Patashnik, *Concrete Mathematics*, Addison-Wesley Publishing Company, 1994.

[3] É. Harangozó, P. Szlávi and L. Zsakó, Joining Programming Theorems, a Practical Approach to Program Building, *in: Annales Universitatis Scientiarum Budapestinensis. Sectio Computatorica* **17** (1998), 155–172.

[4] `http://hu.wikipedia.org/wiki/Permut%C3%A1ci%C3%B3`.

[5] `http://en.wikipedia.org/wiki/Big_O_notation`.

[6] `http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html`.

[7] D. E. Knuth, *The Art of Computer Programming*, Volume 3: Sorting and Searching (2nd Edition), Addison-Wesley Professional, 1998.

[8] J. Nievergelt, J. C. Farrar and E. M. Reingold, *Computer Approaches to Mathematical Problems*, Prentice-Hall, New Jersey, USA, 1977.

[9] OECD, The PISA 2003 assessment framework, Mathematics, reading, science and problem solving knowledge and skills, OECD, Paris, 2003.

[10] P. Szlávi, Mohó algoritmusok módszertana (Methodology of Greedy Algorithms), lecture manuscript, 2003, `http://people.inf.elte.hu/~ szlavi/InfoSavaria05/MohoAlgoritmusok.pdf`.

[11] P. Szlávi, Dinamikus programozás (Dynamic Programming), lecture manuscript, 2007, `http://people.inf.elte.hu/~szlavi/PrM4felev/Pdf/DinamikusProgramozas.pdf`.

[12] P. Szlávi, Dinamikus programozás – Esettanulmányok (Dynamic Programming – Case Studies), lecture manuscript, 2004, `http://people.inf.elte.hu/~szlavi/PrM4felev/DinaProg/DinaPro.pdf`.

[13] P. Szlávi, Formális módszerek a programozásban (Formal Methods in Programming), lecture manuscript, 2003, `http://people.inf.elte.hu/szlavi/PrM4felev/FormModsz.ppt`.

[14] P. Szlávi and L. Zsakó, *Módszeres programozás – Programozási tételek (Systematic Programming – Typical Algorithms of Programming)*, ELTE TTK Informatikai Tanszékcsoport, Budapest, 1994.

[15] P. Szlávi and L. Zsakó, *Programming Versus Application*, in: Informatics Education – The Bridge between Using and Understanding Computers; ISSEP 06 Lecture Notes in Computer Science, 2006, 48–58.

[16] P. Szlávi and L. Zsakó, *Módszeres programozás – Gráfok (Systematic Programming – Graphs)*, ELTE IK, Budapest, 2004.

[17] L. Zsakó, *Módszeres programozás – Hatékonyság (Systematic Programming – Program Efficiency)*, ELTE TTK Informatikai Tanszékcsoport, Budapest, 1991.

[18] L. Zsakó, Variations for Spanning Trees, *in: Annales Mathematicae at Informaticae* **33** (2006), 151–165.

PÉTER SZLÁVI and GÁBOR TÖRLEY
DEPARTMENT OF MEDIA AND EDUCATIONAL INFORMATICS
FACULTY OF INFORMATICS
EÖTVÖS LORÁND UNIVERSITY
BUDAPEST
HUNGARY

E-mail: `szlavip@elte.hu`

E-mail: `pezsgo@elte.hu`

(Received March, 2008)